Python 数据分析 (1st Edition) （Python for Data Analysis）

Chap 6 数据加载、存储与文件格式Data loading, storage, and file formats

内容：

- 多种数据格式文件的加载和存储
- 利用Web API操作网络资源，获取url内容并解析
- 多个表的合并操作merge
- 与数据库的操作

实践：

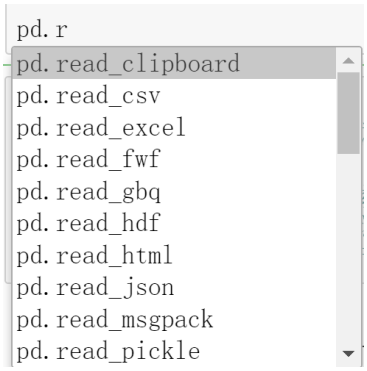
- 表格型格式的数据（csv，table, excel,fwf, clipboard等）
- 层次嵌套格式的数据（JSON，XML，HTML）
- 二进制格式的数据（pickle）
- 数据库（SQL，NoSQL）

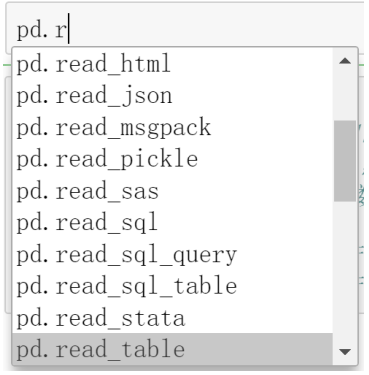
数据要能导入导出Python，这个课程介绍的分析和挖掘技术才有意义。本节课着重介绍多种数据的加载和存储。

In [ ]:
```
# 必要准备工作：导入库，配置环境等
from __future__ import division
import os, sys

# 导入库并为库起个别名
import numpy as np
import pandas as pd
from pandas import Series, DataFrame

# 启动绘图
%matplotlib inline
import matplotlib.pyplot as plt
```

pandas可以读入的数据文件格式包括：

		pandas读入多种数据格式： **- csv, excel** **- html, json** **- pickle** **-** 剪贴板 **-** 数据库**, sql, hdf** **- SASXport**，**Google BigQuery**等





- 通用表格, **table**

1. 读写文本格式的数据（Data in Text Format）

Python在文本和文件处理方面很招人喜欢，原因是其简单的文件交互语法、直观的数据结构，以及诸如元组打包解包之类的便利功能。

pandas提供了一些用于将表格型数据读取为DataFrame对象的函数。表1对它们进行了总结，其中read_csv 和 read_table 是今后用得最多的。

表**1**： **pandas**中对文本格式数据的解析函数

函数	说明
read_csv	从文件、URL、文件型对象中加载带分隔符的数据。默认分隔符为逗号(',')
read_table	从文件、URL、文件型对象中加载带分隔符的数据。默认分隔符为制表符('\t')
read_excel	从文件、URL、文件型对象中加载带分隔符的数据。默认分隔符为逗号(',')
read_fwf	读取定宽列格式数据（也就是说，没有分隔符）
read_clipboard	读取剪贴板中的数据，可以看做read_table的剪贴板。在将网页转换为表格时很有用

这些函数在将文本数据转换为DataFrame时所用到的一些技术，可以划分为以下几个大类：

- 索引：将一个或多个列当做返回的DataFrame处理，以及是否从文件、用户获取列名。
- 类型推断(type inference)和数据转换：包括用户定义值的转换、缺失值标记列表等
- 日期解析： 包括组合功能，比如将分散在多个列中的日期时间信息组合成结果中的单个列
- 迭代： 支持对大文件进行逐块迭代
- 不规整数据问题： 跳过一些行、页脚、注释或其他一些不重要的东西（比如由成千上万个逗号隔开的数值数据）

类型推断（type inference）是这些函数中最重要的功能之一，也就是说，你不需要指定列的类型到底是数组、整数、布尔值，还是字符串。日期和其他自定义类型的处理需要多花点功夫才行。

1. 读取**csv**， **txt**和**table**文件

1. 分隔符

- 以逗号分隔的（csv）文本文件，默认逗号为分隔符，默认第一行进行推理
- 也可以使用read_table来读取csv，但需要指定分隔符为逗号
- 有些表格可能不是用固定的分隔符去分隔字符的（比如空白符或其他模式），对于这种情况，可以编写一个正则表达式来作为read_table的分隔符

- 使用read_table 来读取txt文件，指定正则表达式做分隔符，result = pd.read_table('data/ex3.txt', sep='\s+')

In [ ]:
```
# 执行windows命令，如果是unix，使用cat
!type data\ex1.csv
```

In [ ]:
```
# 也可以使用read_table来读取csv，但需要指定分隔符为逗号，table的默认分隔符为制表符('\t')
pd.read_table('data/ex1.csv', sep=',')
```

In [ ]:
```
# 使用read_csv来读取csv，默认分隔符为逗号
pd.read_csv('data/ex1.csv')
```

不固定分隔符

- 有些表格可能不是用固定的分隔符去分隔字符的（比如空白符或其他模式），对于这种情况，可以编写一个正则表达式来作为read_table的分隔符
- 该文件各个字段由数量不定的空白符分隔，虽然可以对其做一些手工调整，但这个情况还是可以用正则表达式\s+处理比较好，可以如下：

In [ ]:
```
!type data\ex3.txt
# 第一行为tab分隔，第二行以后为空格分隔
```

In [ ]:
```
# **这个例子中，列名比列数量少1， 所以read_table推断第一列应该是DataFrame的索引。**
# 用正则表达式\s+处理分割
result = pd.read_table('data/ex3.txt', sep='\s+')
result
```

2. 标题行

如果本身文件没有标题行，默认header='infer'，则把第一行推理为标题行。

读入没有标题行的文件有两个办法：

- 1）Pandas为其分配默认的列名,默认是从0开始的整数索引
- 2) 自己定义列名,names=['第一季度', '第二季度', '第三季度', '第四季度', '合计']

In [ ]:
```
!type data\ex2.csv
```

In [ ]:
```
# 如果让pandas自动读入数据，默认header='infer'，则把第一行推理为标题行
pd.read_csv('data/ex2.csv')
```

In [ ]:
```
# 如果让pandas为其分配默认的列名,从0开始的整数索引
pd.read_csv('data/ex2.csv', header=None)
```

In [ ]:
```
# 自己定义列名（自己定义names的list） names=['第一季度', '第二季度', '第三季度', '第四季度', '合计']
pd.read_csv('data/ex2.csv', names=['a', 'b', 'c', 'd', 'message'])
```

In [ ]:
```
# 自己定义列名（自己定义names的list） names=['第一季度', '第二季度', '第三季度', '第四季度', '合计']
pd.read_csv('data/ex2.csv', names=['第一季度', '第二季度', '第三季度', '第四季度', '合计'])
```

3. 索引行

- 如果希望将上面的message列做成DataFrame的行索引，可以（1）明确将索引列放到该索引的位置，或（2）通过index_col参数指定“message”列为DataFrame的索引
- 如果希望将多个列做成一个层次化索引，需传入由列编号或列名组成的列表即可

In [ ]:
```
# 如果希望将上面的message列做成DataFrame的行索引，可以（1）指明索引4的位置为DataFrame的索引
names = ['a', 'b', 'c', 'd', 'message']
pd.read_csv('data/ex2.csv', names=names, index_col=[4])
```

In [ ]:
```
# 如果希望将上面的message列做成DataFrame的行索引，可以（2）通过index_col参数指定“message”列为DataFrame的索引
names = ['a', 'b', 'c', 'd', 'message']
pd.read_csv('data/ex2.csv', names=names, index_col='message')
```

In [ ]:
```
!type data\csv_mindex.csv
```

In [ ]:
```
# 如果希望将多个列做成一个层次化索引，只需传入由列编号或列名组成的列表即可：
parsed = pd.read_csv('data/csv_mindex.csv', index_col=['key1', 'key2'])
#parsed = pd.read_csv('data/csv_mindex.csv', index_col=[0,1]) # 作用同上
parsed
```

4. 处理异形格式

这些解析器函数还有许多参数可以帮助处理各种各样的异形文件格式（参看后面的表2）。

- 可以使用 skiprows=[0, 2, 3] 跳过文件的第一行、第三行和第四行
- 可以使用 skipinitialspace=True 跳过分隔符后的空格
- 可以使用 skipfooter=10 跳过文件最后10行
- 默认 skip_blank_lines=True，默认跳过空白行

In [ ]:
```
!type data\ex4.csv
```

In [ ]:
```
# 跳过第一，第三和第四行，忽略最后两行，使用message列做索引，指定 'python' 引擎，因为'c' 引擎不支持 skip_footer
pd.read_csv('data/ex4.csv', skiprows=[0, 2, 3], skipfooter=2, index_col='message', engine='python')
```

5. 缺失值

- 缺失数据经常是（1）没有（空字符串），或（2）用某个标记值表示，默认情况下，pandas会用一组经常出现的标记值进行识别，例如NA、-1.#IND以及NULL等
- isnull 返回布尔值判断是否为缺失值
- na_value可以接受一组用于表示缺失值的字符串，也可以用字典为各列指定不同的NA标记值

In [ ]:
```
!type data\ex5.csv
result = pd.read_csv('data/ex5.csv')
pd.isnull(result)
```

In [ ]:
```
# na_value可以接受一组用于表示缺失值的字符串：
result = pd.read_csv('data/ex5.csv', na_values=['NULL'])
result
```

In [ ]:
```
# 也可以用字典为各列指定不同的NA标记值:
sentinels = {'message': ['foo', 'NULL'], 'something': ['two']}
pd.read_csv('data/ex5.csv', na_values=sentinels)
```

下表列出解析器函数的参数用法

表2： read_csv/read_table函数的参数

参数	说明
path	表示文件系统位置、URL、文件型对象的字符串
sep或delimiter	用于对行中各字段进行拆分的字符序列或正则表达式
header	用作列名的行号。默认为0（第一行），如果没有header行就应该设置为None。即选择哪个行的值作为列名
index_col	用作行索引的列编号或列名。可以是单个名称/数字或由多个名称/数字组成的列表（层次化索引）
names	用于结果的列名列表，结合header=None
skiprows	需要忽略的行数（从文件开始处算起），或需要跳过的行号列表（从0开始）
na_values	一组用于替换NA的值
comment	用于将注释信息从行尾拆分出去的字符（一个或多个）
parse_dates	尝试将数据解析为日期，默认为False。如果为True，则尝试解析所有列。此外，还可以指定需要解析的一组列号或列名。如果列表的元素为列表或元组，就会将多个列组合到一起再进行日期解析工作（例如，日期/时间分别位于两个列中）
keep_date_col	如果连接多列解析日期，则保持参与连接的列。 默认为False
converters	由列号/列名跟函数之间的映射关系组成的字典。例如， {'foo':f}会对foo列的所有值应用函数f
dayfirst	当解析有歧义的日期时，将其看做国际格式（例如，7/6/2012 -> June 7, 2012）。默认为False
date_parser	用于解析日期的函数
nrows	需要读取的行数（从文件开始处算起）
iterator	返回一个TextParser以便逐块读取文件
chunksize	文件块的大小（用于迭代）
skip_footer	需要忽略的行数（从文件末尾处算起）
verbose	打印各种解析器输出信息，比如“非数值列中缺失值的数量”等
encoding	用于unicode的文本编码格式。例如， “utf-8”表示用UTF-8编码的文本
squeeze	如果数据经解析后仅含一列，则返回Series
thousands	千分位分隔符， 如“,”或“.”

6. 处理太大的文件

处理太大的文件，可能需要（1）只读取几行，不需要读取整个文件，或者（2）迭代逐块读取文本文件

- 只读取几行（避免读取整个文件），通过nrows进行指定即可：pd.read_csv('ch06/ex6.csv', nrows=5)
- 需要逐块读取文件，设置chunksize（行数），chunker里面包含很多块，每块的大小为1000行：chunker = pd.read_csv('ch06/ex6.csv', chunksize=1000)
  - read_csv所返回的这个 TextFileReader 对象(以前是TextParser对象),可以根据chunksize对文件进行逐块迭代
  - TextFileReader对象还有一个get_chunk方法，可以读取任意大小的快

In [ ]:
```
import pandas as pd
import numpy as np
from pandas import Series, DataFrame

# ex6 文件采用迭代处理文件，将值计数聚合到“key”列中，即统计每个key对应的四个列的统计量
result = pd.read_csv('data/ex6.csv')
print len(result)
result.head()
```

In [ ]:
```
# (1) 读取大文件的前几行,避免读取整个文件，通过nrows进行指定即可：
pd.read_csv('data/ex6.csv', nrows=3)
```

In [ ]:
```
# （2）逐块读取文件，chunker里面包含很多块，每块的大小为1000行，
# read_csv所返回TextFileReader对象(以前是TextParser对象)
chunker = pd.read_csv('data/ex6.csv', chunksize=1000)
chunker
# TextFileReader对象还有一个get_chunk方法，可以读取任意大小的块
```

In [ ]:
```
# 可以迭代处理文件，将值计数聚合到“key”列中
tot = Series([])
for piece in chunker:
    tot = tot.add(piece['key'].value_counts(), fill_value=0)
tot = tot.sort_values(ascending=False)
tot[:5]
```

In [ ]:
```
tot[:10]
```

7. 将数据写出到文本格式

数据可以被输出为分隔符格式的文本

- 利用DataFrame的to_csv方法，可以将数据写入一个以逗号分隔的文件中
- 还可以使用其他分隔符，如“|”：data.to_csv('data/outsepbar.csv', sep='|')
- 直接写出到sys.stdout，即打印出文本结果：data.to_csv(sys.stdout, sep='|')
- 缺失值在输出结果中被表示为空字符串，使用na_rep参数将其表示为别的标记值：data.to_csv(sys.stdout, na_rep='NULL')
- Series也有一个to_csv方法：

In [ ]:
```
!type data\ex5.csv
# 读入csv
data = pd.read_csv('data/ex5.csv')
```

In [ ]:
```
# 写入csv
data.to_csv('data/out-ex5.csv')
!type data\out-ex5.csv
```

In [ ]:
```
# 使用其他分隔符，如“|”
data.to_csv('data/out-ex5-sepbar.csv', sep='|')
!type data\out-ex5-sepbar.csv
```

In [ ]:
```
# 直接写出到sys.stdout，即打印出文本结果
import os,sys
data.to_csv(sys.stdout, sep='|')
```

In [ ]:
```
# 缺失值在输出结果中被表示为空字符串，使用na_rep参数将其表示为别的标记值：
data.to_csv(sys.stdout, na_rep='NULL')
# 第一个行被认为是header标题行，因此不表示为NULL
# 默认写出行和列的标签
```

In [ ]:
```
# 禁止写出行名和列名， index是行名，header是列名
data.to_csv(sys.stdout, index=False, header=False)
```

In [ ]:
```
# 可以只写出一部分的列，并以指定的顺序排列：
data.to_csv(sys.stdout, index=False, columns=['a','c','b'])
```

In [ ]:
```
# Series也有to_csv方法：
dates = pd.date_range('3/22/2017', periods=5)
ts = Series(np.arange(5.0), index=dates)
ts.to_csv('data/tseries.csv')
!type data\tseries.csv
```

In [ ]:
```
# Series有更为方便的from_csv方法就能将csv文件读取为Series
Series.from_csv('data/tseries.csv', parse_dates=True)
```

8. 手工处理分隔符格式

大部分存储在磁盘上的表格型数据都能用pandas.read_table进行加载。然而，有时还是需要做一些手工处理。由于接收到含有畸形行的文件而使read_table出毛病的情况并不少见。

- 对于任何单字符分隔符文件，可以直接使用Python内置的csv模块。将任意已打开的文件或文件型的对象传给csv.reader

In [ ]:
```
!type data\ex7.csv
```

In [ ]:
```
# 对于任何单字符分隔符文件，可以直接使用Python内置的csv模块
import csv

# 将已打开的文件或文件型的对象传给csv.reader：
#f = open('data/ex7.csv') # 已打开的文件f，将f传给csv.reader亦可
reader = csv.reader(open('data/ex7.csv'))
reader
```

In [ ]:
```
# 对这个reader进行迭代将会为每行产生一个列表（并移除了所有的引号）
for line in reader:
  print line
```

In [ ]:
```
# 为了使数据格式合乎要求，需要做一些整理工作：
lines = list(csv.reader(open('data/ex7.csv'))) # 将reader转换为列表，每个列表中的元素对应一行

header, values = lines[0], lines[1:]
```

```
data_dict = {h: v for h, v in zip(header, zip(*values))}
data_dict
```

2. 读取Microsoft Excel文件

Pandas的 ExcelFile 类支持读取存储在Excel 2003（或更高版本）中的表格型数据。由于ExcelFile用到了xlrd 和 openpyxl 包，需要先安装它们。

xlrd 和 **openpyxl** 两个包 在**Anaconda** 默认安装**!**

- 通过传入一个xls或xlsx文件的路径即可创建一个ExcelFile实例：f=pd.ExcelFile('data/address.xls')
- 通过parse读取到DataFrame中,table是DataFrame类型: table=xls_file.parse('Sheet1')

In [ ]:
```
# 创建一个ExcelFile实例
xls_file = pd.ExcelFile('data/address.xls', encoding='GBK')

# 通过parse读取到DataFrame中,table是DataFrame类型
table = xls_file.parse('Sheet1')
table.head()
```

In [ ]:
```
# 创建一个ExcelFile实例
xls_pub = pd.ExcelFile('data/Pub.xls')
tb1 = xls_pub.parse('Author') # 取Author sheet页
tb1.tail()
```

In [ ]:
```
# 创建一个ExcelFile实例
xls_pub = pd.ExcelFile('data/Pub.xls')
tb2 = xls_pub.parse('Paper') # 取Paper sheet页
tb2.head()
```

In [ ]:
```
# 创建一个ExcelFile实例
xls_pub = pd.ExcelFile('data/Pub.xls')
tb3 = xls_pub.parse('Sheet3') # 取AuthorPaper页
tb3.head()
```

1. 数据集的合并（**merge**）或连接（**join**）

pandas对象中的数据可以通过 pd.merge 根据一个或多个键将不同的DataFrame中的行连接起来，实现的是数据库的连接操作

In [ ]:
```
# 两个表的列名不同，分别进行指定，默认进行inner连接，即键的交集；outer连接求的是键的并集
PAfull = pd.merge(tb3, tb2, left_on='PaperId', right_on='Id')
PAfull[:2]
```

2. 读写层次嵌套格式的数据

1. JSON数据

JSON: JavaScript Object Notation的简称, 成为通过HTTP请求在Web浏览器和其他应用程序之间发送数据的标准格式之一。

- import json

- 是一种比表格型文本格式（如CSV）更灵活的数据格式; 除了空值null和一些其他细微差别（如列表末尾不允许存在多余的逗号）之外，JSON非常接近于有效的Python代码。
- 基本类型有对象（字典）、数组（列表）、字符串、数值、布尔值以及null。对象中所有的键都必须是字符串。许多Python库都可以读写JSON数据，在Python标准库中的构建包是json。通过json.loads即可将JSON字符串转换成Python形式

In [ ]:
```
import json # 标准Python库json
```

In [ ]:
```
json_string = '{"name":"USA","state":[{"nm":"Washington","cities":[{"city":["S", "O"]}]},{"nm":"Calif","cities":[{"city":["A", "B"]}]}]}'
```

In [ ]:
```
# 通过json.loads即可将JSON字符串转换成Python形式:
result = json.loads(json_string)
result
```

In [ ]:
```
# 相反，json.dumps则将Python对象转化成JSON格式:
asjson = json.dumps(result)
print asjson
```

如何将（一个或一组）**JOSN**对象转换为**DataFrame**或其他便于分析的数据结构？

最简单的方式是：向DataFrame构造器传入一组JSON对象，并选取数据字段。

In [ ]:
```
cities = DataFrame(result['state'], columns=['cities'])
cities
```

Tweets推文以**JSON**格式存储

提取推文内容进行解析

In [ ]:
```
tweets = pd.read_json(open('data/tweets.json'))
print len(tweets)

tweets[:2]
# 直接读入json文件的解析不好，没有区分层次结构
```

In [ ]:
```
# 使用python的json库
import json
with open('data/tweets.json') as json_file:
    tweets = json.load(json_file)
len(tweets) # 有281条推文
```

In [ ]:
```
print len(tweets[0]) # 每条推文有32个属性
#tweets[0] # 显示第一条推文的内容
```

In [ ]:
```
print tweets[0]['id'] # 推文编号
print tweets[0]['text'] # 推文内容
print tweets[0]['retweet_count'] # 推文转发次数
print tweets[0]['favorite_count'] # 推文被赞次数
print tweets[0]['user']['name'] # 作者的名字
print tweets[0]['user']['favourites_count'] # 该作者（历史以来）点赞推文次数
print tweets[0]['user']['friends_count'] # 该作者的好友个数
print tweets[0]['user']['followers_count'] # 该作者有多少追随者
```

In [ ]:
```
print tweets[5]['id'] # 推文编号
print tweets[5]['text'] # 推文内容
print tweets[5]['retweet_count'] # 推文转发次数
print tweets[5]['favorite_count'] # 推文被赞次数
print tweets[5]['user']['name'] # 作者的名字
print tweets[5]['user']['favourites_count'] # 该作者（历史以来）点赞推文次数
print tweets[5]['user']['friends_count'] # 该作者的好友个数
print tweets[5]['user']['followers_count'] # 该作者有多少追随者
```

** 许多网站都有一些通过JSON或其他格式提供数据的公共API。通过Python访问这些API的办法有不少。一个简单易用的办法（推荐办法）是requests包。

例如，为了在Twitter上搜索“trump”，可以发送一个HTTP GET请求，如下所示：

```
# 发送一个HTTP GET请求
import requests

url = 'http://search.twitter.com/search.json?q=trump'
resp = requests.get(url)

# Response对象的text属性含有GET请求的内容。许多Web API返回的都是JSON字符串，我们必须将其加载到一个Python对象中：

# JSON解析
import json
data = json.loads(resp.text)

# 结果转换为DataFrame格式：
results = DataFrame(data)
results
```

2. HTML 和 XML 格式

许多网站都将数据放到HTML表格中以便在浏览器中查看，但不能以一种更易于机器阅读的格式（如JSON、HTML或XML）进行下载。

Yahoo!Finance的股票期权数据就是这样。期权是指使你有权从现在开始到未来某个时间（到期日）内以某个特定价格（执行价）买进（看涨期权）或卖出（看跌期权）某公司股票的衍生合约。人们的看涨和看跌期权交易有多种执行价和到期日，这些数据都可以在Yahoo!Finance的各种表格中找到。

Python有很多可以读写HTML和XML格式数据的库。lxml能够高效且可靠地解析大文件,有多个编程接口。首先，我们可以使用lxml.html处理HTML，然后再用lxml.objectify做一些XML处理。

直接从**Yahoo!Finance** 的 **Web** 收集股票数据。

1. HTML格式

- 首先，找到获取数据的URL，利用urllib2将其打开，然后用lxml解析得到的数据流，如下：

In [ ]:
```
from lxml.html import parse
from urllib2 import urlopen
```

```
parsed = parse(urlopen('http://finance.yahoo.com/q/op?s=AAPL+Options'))
doc = parsed.getroot()
```

- 通过doc这个对象，可以获取特定类型的所有HTML标签（tag），比如含有所需数据的table标签。给这个简单的例子加点启发性，假设你想得到该文档中的所有的URL链接。？HTML中的链接是a标签。使用文档根节点的findall方法以及一个XPath（对文档的“查询”的一种表示手段）：

In [ ]:
```
links = doc.findall('.//a')
links[15:20]
```

- 但这些是表示HTML元素的对象。要得到URL和链接文本，必须使用各对象的get方法（针对URL）和text_content方法（针对显示文本）：

In [ ]:
```
lnk = links[28]
```

In [ ]:
```
lnk.get('href')
```

In [ ]:
```
lnk.text_content()
```

因此，编写下面的这条列表推导式（**list comprehension**）即可获取文档中的全部**URL**：

In [ ]:
```
urls = [lnk.get('href') for lnk in doc.findall('.//a')]
urls[-10:]
```

- 现在，从文档中找出正确表格的方法就是反复试验了。有些网站会给目标表格假设一个id属性。
- 现在，把所有步骤都结合起来，就可以将数据转换为一个DataFrame。由于数值型数据仍然是字符串格式，所以我们希望将部分列（可能不是全部）转换为浮点数格式。虽然可以手工实现该功能，但是pandas恰好就有一个TextParser类可以用于自动类型转换（read_csv和其他解析函数其实在内部就用到了它）：
- 最后，对那两个lxml表格对象调用该解析函数并得到最终的DataFrame：

2. XML解析

XML（Entensible Markup Language）是另一种常见的支持分层、嵌套数据以及元数据的结构化数据结构格式。

用于解析XML数据的接口很多。

In [ ]:
```
!type data\xml-sample.xml
```

In [ ]:
```
# 安装lxml模块后需要导入lxml模块的etree类
from lxml import etree

# 解析xml文件
xml_file = etree.parse("data/xml-sample.xml")

# 获取文件的根节点
root_node = xml_file.getroot()
```

In [ ]:
```
# 使用for遍历节点的子节点
for sub_node in root_node:
  # 获取节点的标签和内容
  print sub_node.tag, " : ", sub_node.text
```

3. 二进制数据

1. pickle序列化

实现数据的二进制格式存储最简单的办法之一是使用Python内置的pickle序列化。

pandas对象都有一个用于将数据以pickle形式保存到磁盘上的save方法：

In [ ]:
```
df = pd.read_csv('data/ex1.csv')
df
```

In [ ]:
```
# to_pickle进行二进制存储，把数据写入pickle
df.to_pickle('data/ex1_pickle')
```

In [ ]:
```
# Pandas的read_pickle函数将数据读回到Python：
pd.read_pickle('data/ex1_pickle')
```

注意：**pickle**仅建议用于短期存储格式。其原因是很难保证该格式永远是稳定的。今天的**pickle**对象可能无法被后续版本的库**unpickle**处理。

2. HDF5格式

很多工具都能实现高效读写磁盘上以二进制格式存储的科学数据。HDF5就是其中一个流行的工业级库，它是一个C库，带有许多语言的接口，如Java、Python和MATLAB等。HDF5中的HDF指的是层次型数据格式（hierarchical data format）。每个HDF5文件都含有一个文件系统式的节点结构，它使你能够存储多个数据集病支持元数据。与其他简单格式相比，HDF5支持多种压缩器的即时压缩，还能更高效地存储重复模式数据。对于那些非常大的无法直接放入内存的数据集，HDF5就是不错的选择，因为它可以高效地分块读写。

Python中的HDF5库有两个接口（即PyTables和h5py），它们各自采取了不同的问题解决方式。h5py提供了一种直接而高效的HDF5 API房屋接口，而PyTables则抽象了HDF5的许多细节以提供多种灵活的数据容器、表索引、查询功能以及对核外计算技术（out-of-core computation）的某些支持。

pandas有一个最小化的类似于字典的HDFStore类，它通过PyTables存储pandas对象：

pip install PyTables 没有安装成功

使用pip install tables， 是默认安装

h5py已经是默认标准安装

In [ ]:
```
store = pd.HDFStore('data/mydata.h5')
df = pd.read_csv('data/ex1.csv')
store['obj1'] = df
store['obj1_col'] = df['a']
store
```

- HDF5文件中的对象可以通过与字典一样的方式进行获取：

In [ ]:
```
# 即frame
store['obj1']
```

In [ ]:
```
# 即一个列series
store['obj1_col']
```

In [ ]:
```
store.close()
#os.remove('data/mydata.h5') # 删去hdf5数据
```

注意：

如果需要处理海量数据，建议好好研究一下**PyTables**和**h5py**，看看它们是否能满足你的哪些需求。由于许多数据分析问题都是**IO**密集型（而不是**CPU**密集型），利用**HDF5**这样的工具能显著提升应用程序的效率。

HDF5 不是数据库。它最适合用作**“一次写多次读”**的数据集。虽然数据可以在任何时候被添加到文件中，但如果同时发生多个写操作，文件就可能会被破坏。

4. 使用数据库

1. SQL数据库

在许多应用中，数据很少取自文本文件，因为用这种方式存储大量数据很低效。基于SQL的关系型数据库（如SQL Server、PostgreSQL和MySQL等）使用非常广泛，此外还有一些非SQL（即所谓的NoSQL）型数据库也变得非常流行。数据库的选择通常取决于性能、数据完整性以及应用程序的伸缩性需求。

将数据从SQL加载到DataFrame的过程很简单，此外pandas还有一些能够简化该过程的函数。例如，可以使用一款嵌入式SQLite数据库（通过Python内置的sqlite3驱动器）

In [ ]:
```
import sqlite3  # 默认已经安装
query = """
CREATE TABLE test
(a VARCHAR(20), b VARCHAR(20),
 c REAL,        d INTEGER
);"""
# 使用 ":memory:" 打开一个驻内存的数据库，而不是驻磁盘的数据库
con = sqlite3.connect(':memory:')
con.execute(query)
con.commit()
```

In [ ]:
```
# 插入几行数据
data = [('Atlanta', 'Georgia', 1.25, 6),
        ('Tallahassee', 'Florida', 2.6, 3),
        ('Sacramento', 'California', 1.7, 5)]
stmt = "INSERT INTO test VALUES(?, ?, ?, ?)"
con.executemany(stmt, data)
con.commit()
```

In [ ]:
```
#从表中选取数据时，大部分Python SQL驱动器（PyODBC、psycopg2、MySQLdb、pymssql等）都会返回一个元组列表rows：
cursor = con.execute('select * from test')
```

```
rows = cursor.fetchall()
rows
```

In [ ]:
```
# 可以将这个元组列表传给DataFrame的构造器，但还需要列名（位于游标的description属性中）：
cursor.description
```

In [ ]:
```
cursor.description[1][0]
```

In [ ]:
```
DataFrame(rows, columns=zip(*cursor.description)[0])
```

In [ ]:
```
import pandas.io.sql as sql
sql.read_sql('select * from test', con)
```

- 这种数据规整操作相当多，你肯定不想每查一次数据库就重写一次。pandas有一个可以简化该过程的read_frame函数（位于pandas.io.sql模块）。只需传入select语句和连接对象即可：

2. NoSQL数据库：MongoDB

NoSQL数据库有许多不同的形式。有些是简单的字典式键值对存储（如BerkeleyDB和Tokyo Cabinet），另一项则是基于文档的（其中的基本单元是字典型的对象）。这里我们选择的是MongoDB。我们需要现在自己的电脑上启动一个MongoDB实例，然后用pymongo（MongoDB的官方驱动器）通过默认端口进行连接：

In [ ]:
```
from pymongo import MongoClient
client = MongoClient()
# client = MongoClient('localhost', 27017)
#client = MongoClient('mongodb://localhost:27017/')
```

In [ ]:
```
# 建立一个Mongo DB
db = client['mydatabase']
```

In [ ]:
```
collection = db['mycollect']
```

- 存储在MongoDB中的文档被组织在数据库的表中。MongoDB服务器的每个运行实例可以有多个数据库，而每个数据库又可以有多个集合。

In [ ]:

In [ ]:

In [ ]: